Monads Explained

Izaak Weiss

What are Monads?

It is entirely reasonable that this is the first question that anyone learning Monads asks, and it is also entirely reasonable that anyone who is teaching Monads tries to answer it. However, Monads are a complex concept that cannot be explained in a single sentence or even a single paragraph; to understand monads you must simultaneously understand the problem they are trying to solve, their implementation, the interface for working with them, and the theoretical computational background. Therefore, I will not try and answer this question in a single phrase; my explanation of what Monads are is the entirety of this paper.

Our First Monads

In this section, we'll explore a common problem surrounding how to report errors to the caller of a function.

Division

```
def division(x, y):
    return x / y
```

Consider the above code fragment. This is a very simple function; one that is so simple it hardly deserves to exist. However, this function, despite it's simplicity, can throw an error if it's inputs are badly formed. `division(1.0,0.0)` will throw a `ZeroDivisionError: division by zero`. It's possible that we want to recover from this error gracefully; check the inputs and see if an error will occur, and return some error code instead of raising an exception.

However, we must decide what the error code should be. We can't choose `0.0`, because that is correctly returned by `division(0.0,1.0)`. We can't choose `-1.0`, because that is correctly returned by `division(-1.0,1.0)`. In fact, this function can return any possible floating point number, so we can't choose a

floating point number as our error code. Luckily, python is dynamically typed, so we can return a float on success, but `None` on failure.

```
def division(x, y):
    if y == 0:
        return None
    return x / y
```

Now we've written a function that checks whether or not division is possible, and performs division if it is, but returns an error code if it is not.

Indexing

```
def index(ls, i):
    return ls[i]
```

The above code is similar to the last example; it will perform an index lookup into a list, and return the item from the list if it can. If it can't, we're still left with the problem that it throws an exception. `index([1,2,3],10)` throws an `IndexError: list index out of range`. Let's try and do the same thing as above; rewrite this function so that it doesn't throw an error, but instead uses an error code to signal something has gone wrong.

Our first guess might be to have our error code be the same as above, and just return `None`. However, this isn't possible, because the code `index([None], 0)` would also return `None`, making it impossible to tell in practice if we have an error or not. In fact, python lets any value be inside of a list; there is no possible error code we can return that can't also be in the list. Luckily again, python lets us use multiple return values.

```
def index(ls, i):
    if i < 0 or i >= len(ls):
        return True, None
    return False, ls[i]
```

This allows us to actually check whether or not this function has failed, without worrying about receiving an exception. This allows us to handle errors from outside of the function in a logical way:

```
failed, value = index([1,2,3],0)
if failed:
    print("Oh no, we failed")
else:
    print(value)
```

Combining The Above

```
def divide_elements(ls, i1, i2):
    return ls[i1]/ls[i2]
```

Now, we've combined the two operations in python which might lead to an exception, and we've done it in a way that allows for three different operations to result in an exception. We can use the above functions to rewrite this function so that it won't throw any errors, by checking after each step for an error code.

```
def index(ls, i):
    if i < 0 or i >= len(ls):
        return True, None
    return False, ls[i]

def division(x, y):
    if y == 0:
        return None
    return x / y

def divide_elements(ls, i1, i2):
    failure, v1 = index(ls, i1)
    if failure:
        return None

    failure, v2 = index(ls, i2)
    if failure:
        return None

    return division(v1, v2)
```

This code works nicely; you can throw two types of errors at it, and it returns `None` when either error would occur. `divide_elements([1,2,3,0],1,10)` and `divide_elements([1,2,3,0],1,3)` both return `None`, and `divide_elements([1,2,3,0],1,2)` returns `0.666666`.

This is nothing special so far. All we've done is rewritten a piece of code so that it doesn't throw an exception. In fact, we've probably made the code worse; python is built around dynamic types and exception throwing, so the more pythonic way of writing the above code is more like the following.

```
def divide_elements(ls, i1, i2):
    try:
        return ls[i1]/ls[i2]
    except (IndexError, ZeroDivisionError):
        return None
```

However, consider these problems in a language like C. No exceptions, no dynamic

return types, and no returning multiple values. This isn't to say the problem can't be solved. C programmers have their own ways of getting around this sort of problem. But different languages have different requirements, and there are many ways to solve a problem. Next up I'm going to introduce our first monad, a monad that helps deal with the problem we faced above - it's not the only way, but it's an elegant way that works in statically typed languages without exceptions.

Our First Monad

The Option Monad, also called the Maybe Monad in many programming languages, is a way of representing the result of a function, the result of any computation, that might result in an error and produce no meaningful output. We call an Option Monad that has a value and represents a successful computation Some, and we call one that doesn't have a value and represents a failed computation None.

These are really easy to write in programming languages like Haskell, Scala, or Rust, but I'm going to write it in Python to help avoid any confusion about the inner workings of the Monad. In the end, all Monads are just objects, like trees, lists, or dictionaries.

```
class OptionException(Exception):
    pass

class Option:
    def __init__(self, failed, value):
        self._failed = failed
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Option.none()'
        else:
            return 'Option.some({})'.format(self._value)

    def __str__(self):
        if self._failed:
            return 'None'
        else:
            return 'Some({})'.format(self._value)

    def is_some(self):
        if self._failed:
            return False
        return True
```

```
    def is_none(self):
        return not self.is_some()

    def unwrap(self):
        if self._failed:
            raise OptionException('This Option has no value')
        else:
            return self._value

    @classmethod
    def some(cls, x):
        return cls(False, x)

    @classmethod
    def none(cls):
        return cls(True, None)
```

This is the longest piece of code we've had so far, so let me break it down bit by bit.

```
class OptionException(Exception):
    pass
```

This is how you create a custom exception in python. I could have used a built in error if I wanted, but I wanted it to be clear that this was part of the Option Monad that was causing the exception and not something else. You probably should object at this point, and yell at me, saying something like, "isn't the whole point of the Option monad to get rid of exceptions?" Well, if you use the Option Monad correctly, you'll never run into this exception.

```
class Option:
    def __init__(self, failed, value):
        self._failed = failed
        self._value = value
```

The `__init__` function is the constructor or initializer function for classes in python. Here all we do is create a boolean that indicates whether or not we have failed the computation or not, and if we haven't failed it, we store the result of the computation in `_value`. Note that if `_failed` is `True`, then we don't care what is in `_value`, because the computation has failed and that value has no meaning. I will note that the users of this class will probably never call `__init__` themselves, as we will later write alternate constructors that are easier for people to use.

```
    def __repr__(self):
        if self._failed:
            return 'Option.none()'
        else:
```

```
            return 'Option.some({})'.format(self._value)

    def __str__(self):
        if self._failed:
            return 'None'
        else:
            return 'Some({})'.format(self._value)
```

This is how you define string representations of values in python. `__str__` is called when you cast something to a string or print it, and `__repr__` is shown in the python REPL, and is supposed to be a little more informative. But these are mostly for ease of use; they don't really matter when it comes to Monads in general.

```
    def is_some(self):
        if self._failed:
            return False
        return True

    def is_none(self):
        return not self.is_some()

    def unwrap(self):
        if self._failed:
            raise OptionException('This Option has no value')
        else:
            return self._value
```

These three functions are the meat of the Option Monad; these are the ways we interact with it. The `is_some` function returns `True` when there is a meaningful return value, and `False` if the computation failed. `is_none` does the opposite. `unwrap` returns the value of the Option Monad *if there is a value to be returned*, otherwise it throws an error. In order to use the `unwrap` function without an error, you must first check and see whether the computation succeeded or failed.

```
    @classmethod
    def some(cls, x):
        return cls(False, x)

    @classmethod
    def none(cls):
        return cls(True, None)
```

These functions, decorated with `@classmethod`, aren't methods of the object. Instead, they're methods that exist as part of the class itself; here, we use them as alternate constructors.

At this point, let me rewrite our exception-free code from above using the Option Monad.

```
def division(x, y):
    if y == 0:
        return Option.none()
    return Option.some(x / y)

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def divide_elements(ls, i1, i2):
    res1 = index(ls, i1)
    if res1.is_none():
        return Option.none()

    res2 = index(ls, i2)
    if res2.is_none():
        return Option.none()

    return division(res1.unwrap(), res2.unwrap())
```

The above code is the exact same length in lines; and already has some benefits. One, these functions have a return type that can be determined just be looking at the code - not as useful in python as in other languages, but I hope you can see why they'd be useful in statically typed languages. Two, we don't have to remember the convention for every function. Before, we had to remember that `division` returned `None` for an error, but `index` returned `False, None` for an error. Despite these benefits, it's not that much of an improvement.

And that's because we haven't yet implemented the most important function for a Monad. This is the most important but also the most complicated part of the Option Monad, so I am going to insert a header for this.

The Bind Function

```
    def bind(self, function):
        if self.is_none():
            return type(self).none()

        val = self.unwrap()
        return function(val)
```

The bind function first checks whether or not `self` is None or Some. If `self` is None, then we return a new None instance. If self is Some, we unwrap the value, and pass it into the function, returning what the function returns. Because we're sort of pretending to be in a typed language here, we should remember that the

function passed in *should* return an Option Monad.

So, with `bind` in mind, let's consider the our code again.

```
def division(x, y):
    if y == 0:
        return Option.none()
    return Option.some(x / y)

def index(ls, i):
    if i < 0 or i >= len(ls):
        return Option.none()
    return Option.some(ls[i])

def divide_elements(ls, i1, i2):
    res1 = index(ls, i1)
    res2 = index(ls, i2)

    partial = lambda x: lambda y: division(x,y)

    return res2.bind(res1.bind(partial))
```

I've changed a few things, so go back and look at what I've done. There's this new line `partial = lambda x: lambda y: division(x,y)`. This is a way of defining an inline function in python; here, I have a function of x that returns a function of y, that itself returns `division(x,y)`. That's a bit complicated, so think about it for a second.

What I then do on the return statement line is I use bind to apply the function to these arguments; the function `division` will fail drastically if it gets an Option Monad in as an argument; it would fail on the comparison to zero and it would also fail on trying to divide two Option Monads. By using `bind`, I am telling the Options to apply the function to themselves if they have a value, or returning None if they don't have a value. This wraps the error handling into the data structure that represents errors without me having to ever write an explicit `if x.is_some():` check.

You may still think this sort of thing is useless; and in python, for such a simple example, it kinda is! But as we continue to explore Monads, we will encounter some weirder and wilder examples of things that Monads make simpler.

A More Complex Example

In order to give a more illustrative example of where the Option Monad can be more useful, consider the following problem; open a file, and read the first whitespace separated word from the beginning of the file, and parse it into an integer if possible. This problem is fairly easy to do with built in python

functions, but the Option monad can make error handling easier. However, none of python's built in functions use the Option monad, so we will have to rewrite them so that they do. In languages with the Option monad as a star player, this isn't an issue.

```
def option_open(filename, mode='r'):
    try:
        fd = Option.some(open(filename, mode=mode))
    except Exception:
        fd = Option.none()
    return fd

def option_read(fd, size=-1):
    try:
        data = Option.some(fd.read(size))
    except Exception:
        data = Option.none()
    return data

import re

def option_match(pattern, string):
    match = re.match(pattern, string)
    if match:
        match = Option.some(match)
    else:
        match = Option.none()
    return match

def option_get_group(match, group):
    try:
        g = match.group(group)
    except Exception:
        g = None

    if g == None:
        g = Option.none()
    else:
        g = Option.some(g)

    return g

def option_int(s):
    try:
        i = Option.some(int(s))
    except Exception:
```

```
        i = Option.none()

    return i
```

I'm not going to explain these functions too much; basically they do the same thing as their built in python counterparts, but they return Some if the computation succeeds and None if it fails, instead of throwing an error or using some other return code. This will allow us to use `bind` to chain these functions.

```
result = (Option.some('text.txt')
.bind(option_open)
.bind(option_read)
.bind(lambda x: option_match(r'\s*(\S*)', x))
.bind(lambda x: option_get_group(x, 1))
.bind(option_int))
```

This code is useful because if an error happens at any time during the computation, it just passes a None Option through the rest of the bind functions.

We can make this look cooler by choosing an infix operator to overload. By convention, `>>=` is used, but that's hard to to do in python, so we are going to use `>>`. We can override that operator in python with the following code:

```
def __rshift__(self, function):
    return self.bind(function)
```

This code let's us rewrite the above option code as the following.

```
result = (
    Option.some('text.txt')
      >> option_open
      >> option_read
      >> (lambda x: option_match(r'\s*(\S*)', x))
      >> (lambda x: option_get_group(x, 1))
      >> option_int
    )
```

The Result Monad

There is one downside to the Option Monad as far as we've discussed it. When the above code is run, you can't tell where or why an error occurs; no matter where the error occurs, you just end up with an inscrutable None.

Luckily, these is a simple modification to the Option Monad that makes is able to convey error messages. Some languages call this the Result Monad. I'm not going to go in depth as to how it works, because it's very similar to the Option Monad. I will, however, highlight one function from below.

```
    def recover(self, function):
        if self.is_error():
            return function()

        return self
```

The `recover` function is the opposite of `bind`; if the result is a failure, it will attempt to recover computation by replacing the current Result Monad with the result of whatever function is passed in. If the result is a success, however, it leaves the value alone.

Here is the full source code for the Result Monad; I do recommend familiarizing yourself with it before we move on.

```
class ResultException(Exception):
    pass

class Result:
    def __init__(self, failed, value, message):
        self._failed = failed
        self._message = message
        self._value = value

    def __repr__(self):
        if self._failed:
            return 'Option.error({})'.format(repr(self._message))
        else:
            return 'Option.ok({})'.format(repr(self._value))

    def __str__(self):
        if self._failed:
            return 'Error({})'.format(self._message)
        else:
            return 'Ok({})'.format(self._value)

    def is_ok(self):
        if self._failed:
            return False
        return True

    def is_error(self):
        return not self.is_ok()

    def unwrap(self):
        if self.is_ok():
            return self._value
        else:
```

```
            raise ResultException('This Result is an Error')

    def error_msg(self):
        if self.is_error():
            return self._message
        else:
            raise ResultException('This Result is Ok')

    def bind(self, function):
        if self.is_error():
            return self

        val = self.unwrap()
        return function(val)

    def recover(self, function):
        if self.is_error():
            return function()

        return self

    def __rshift__(self, function):
        return self.bind(function)

    @classmethod
    def ok(cls, val):
        return cls(False, val, None)

    @classmethod
    def error(cls, msg):
        return cls(True, None, msg)

# The following are built in functions rewritten to work with the Result Monad

def result_open(filename, mode='r'):
    try:
        fd = Result.ok(open(filename, mode=mode))
    except Exception:
        fd = Result.error("Failed to open the file")
    return fd

def result_read(fd, size=-1):
    try:
        data = Result.ok(fd.read(size))
    except Exception:
        data = Result.error("Failed to read from the file")
```

```
    return data

import re

def result_match(pattern, string):
    match = re.match(pattern, string)
    if match:
        match = Result.ok(match)
    else:
        match = Result.error("Failed to match the pattern")
    return match

def result_get_group(match, group):
    try:
        g = match.group(group)
    except Exception:
        g = None

    if g == None:
        g = Result.error("Failed to get the group from the match")
    else:
        g = Result.ok(g)

    return g

def result_int(s):
    try:
        i = Result.ok(int(s))
    except Exception:
        i = Result.error("Failed to parse into an integer")
    return i


result = (
    Result.ok('text.txt')
      >> result_open
      >> result_read
      >> (lambda x: result_match(r'\s*(\S*)', x))
      >> (lambda x: result_get_group(x, 1))
      >> result_int
    )

print(result)
```

Either Monad

Another common monad that is not worth writing in python but invaluable in some other languages, and is in fact a more general form of the Result Monad, is the Either Monad. An Either Monad is really useful in strictly typed languages, because it can be returned from a function that can return two possible types.

I'm not going to implement this in python, because this is mostly useless for a python program, but it's essentially identical to the Result Monad implementation. Try it out!

A Parsing Monad

This is our first look into a complex and powerful Monad that represents a very useful computer science tool. We are going to use a Monad called the Parsing Combinator, which basically recreates and improves upon regex using Monads. Here we see the truth of the oft said but little understood aphorism that Monads represent computation; here, they represent the computation of regular expressions.

It's also worth noting that there are many python Parsing Combinator libraries on the python package index, so if you want to use this for real, find one of them, and not mine; theirs is better written, faster, and provides more functionality.

The Code

First of all, I am going to use the above Result Monad to help me implement this Parsing Combinator. Even though the parsing itself is Monadic, it has to return a result that can either be a success or a failure; so the Result Monad is going to be useful.

With that said, let's step through the code for our Parsing Combinator.

```
class Parser:
    def __init__(self, function):
        self._function = function

    def __call__(self, text, index):
        return self._function(text, index)

    def __repr__(self):
        return '<Parsing Combinator>'
```

The basic idea behind our Parsing Combinator is that the Monad holds a function that takes a string, which we're going to call `text`, and it returns a Result Monad. If the result is a success, the Result Monad holds a tuple of the text that has

matched the current parser, and the remainder of `text`. In order to make this a bit easier for us, we will implement the special method `__call__`, so that we can write stuff like `self(text)` instead of `self._function(text)` in the future. It's really hard to print functions meaningfully, so we're just going to define our representation as a constant string.

I'm going to slightly deviate from our already established order to introduce the constructors first; then we can deal with the ways we can bind to and combine Parsing Combinators. These constructors will do two things; define a function, and then create a Parser using that function.

```
    @classmethod
    def char(cls, val):

        def match_char(text):
            try:
                current = text[0]
            except IndexError:
                return Result.error('End of String encountered, but '
                    + '{} is still expected'.format(repr(val)))

            if current == val:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Failed to match character {} at {}'
                    .format(repr(val), repr(text)))

        return Parser(match_char)

    @classmethod
    def empty(cls):

        def match_empty(text):
            return Result.ok(('', text))

        return Parser(match_empty)

    @classmethod
    def oneof(cls, charls):

        def match_charls(text):
            try:
                current = text[0]
            except IndexError:
                return Result.error('End of String encountered, but one of '
                    + '{} is still expected'.format(list(charls)))
```

```
            if current in charls:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Failed to match one of {} at {}'
                    .format(list(charls), repr(text)))

        return Parser(match_charls)

    @classmethod
    def noneof(cls, charls):

        def none_charls(text):
            try:
                current = text[0]
            except IndexError:
                return Result.error('End of String encountered, but none of '
                    + '{} is still expected'.format(repr(text)))

            if current not in charls:
                return Result.ok((text[0], text[1:]))
            else:
                return Result.error('Found one of {} at {} when'
                    + 'there should be none of'.format(list(charls), repr(text)))

        return Parser(none_charls)
```

These all follow fairly simple rules; they are the equivalent of some fairly basic regular expressions. The first one tries to compare the first character of the text to a character, and if it can, it pulls that character off of the text, and returns the character and the remainder as the result. The second one does absolutely nothing to the input text, simply returning the empty string and the input unchanged. The third and fourth represent character classes, but still match exactly one character from the beginning of the text. To do advanced parsing, we're going to need some more advanced constructors; ones that combine other parsers into new parsers.

```
    def combine(self, other, function):

        def combine_func(match, rest):
            res = other(rest)
            if res.is_ok():
                other_match, rest = res.unwrap()
                new_match = function(match, other_match)
                return Result.ok((new_match, rest))
            else:
                return res
```

```
        return Parser(lambda text: self(text)
            .bind(lambda res: combine_func(*res)))
```

`combine` is the most useful function for us; it takes two parsers (`self` and `other`) and a function. It creates a new parser that first executes `self` on the input text, and then it executes `other` on the remaining text after `self`'s match. Then it uses `function` to combine the two matches, and returns that combination along with the remaining text. We're also going to define a few functions that call combine with a particular function

```
    def concat(self, other):
        return self.combine(other, lambda x,y: x + y)

    def first(self, other):
        return self.combine(other, lambda x,y: x)

    def last(self, other):
        return self.combine(other, lambda x,y: y)

    def tuple(self, other):
        return self.combine(other, lambda x,y: (x,y))
```

These functions are the most common choices for what you might want to use to combine the two matches. Most common is `concat`, which simply concatenates the two strings; this is the default regex behavior. Then there is `first` and `last`, which instead of combining the two values, instead discard one of the values. We will see later why this is useful. Finally, we can put the two values into a tuple instead of concatenating them. This is mostly useful when used in conjunction with `bind`, which we will discuss later. However, I've provided a few examples of working with these functions.

```
match_a = Parser.char('a')
match_b = Parser.char('b')

# This will match the string 'ab' and return the string 'ab'
match_a.concat(match_b)

# This will match the string 'ab' and return the string 'a'
match_a.first(match_b)

# This will match the string 'ab' and return the string 'b'
match_a.last(match_b)

# This will match the string 'ab' and return the tuple ('a','b')
match_a.tuple(match_b)
```

We also have a few other important functions that we use to build more complex parsers.

```
    def choice(self,other):

        def choice_func(text):
            return self(text).recover(lambda: other(text))

        return Parser(choice_func)
```

`choice` implements the ability to try one parser, and if it fails, recover by trying another. Be careful with this one, because it only operates locally. If you try and match **x** or **y**, and **x** succeeds in matching the text but puts you in a corner that causes failure later on, it won't backtrack and try **y**. It will only backtrack and try **y** if matching **x** fails.

```
    def many(self, function=lambda x,y: x + y):

        def repeat_func(text):
            res = self(text)

            if res.is_error():
                return Result.ok(('',text))

            match = res.unwrap()[0]
            rest = res.unwrap()[1]

            res = self(rest)

            while res.is_ok():
                match = function(match, res.unwrap()[0])
                rest = res.unwrap()[1]
                res = self(rest)

            return Result.ok((match, rest))

        return Parser(repeat_func)

    def many1(self, function=lambda x,y: x + y):
        return self.combine(self.many(function), function)
```

`many` is perhaps the most complicated constructor. What we want to do is continually match one parser, using a function to combine the results, until a failure occurs; but when a failure occurs, we want to ignore the failure and return that previous match. We also have a useful `many1` function, which is the same as `many` but demands that at least one match succeeds.

```
    def optional(self):
        return self.choice(Parser.empty())
```

Finally, `optional` tries to match the input text with `self`, but if it fails, it

matches nothing and finishes.

Finally, we add symbolic versions of many of the above functions. This is purely for ease of reading the expressions we will write; you will see that they can get pretty complex, and we don't want to

```
    # |
    def __or__(self, other):
        return self.choice(other)

    # +
    def __add__(self, other):
        return self.concat(other)

    # =>
    def __ge__(self, other):
        return self.last(other)

    # <=
    def __le__(self, other):
        return self.first(other)

    # &
    def __and__(self, other):
        return self.tuple(other)
```

Now we can move on to the Monadic functions.

```
    def bind(self, function):

        def bind_func(result):
            function(result[0]).bind(lambda x: Result.ok(x, result[1]))

        return Parser(lambda text: self(text).bind(bind_func))
```

`bind` is, of course, the star of the show. Lets break this down into steps, starting from the the return statement and moving back.

- We're creating a new Parser, so we're passing in a function that takes text, and should return a Result Monad holding `(matched value, remainder of text)`.
- The first thing this function does is pass `text` through the current parser's function. This should result in a Result Monad holding `(matched value, remainder of text)`.
- Then, we call bind on that result, passing in `bind_func`. `bind_func` is going to get a tuple as it's only argument, and should return a Result Monad holding `(matched value, remainder of text)`.
- `bind_func` takes the result, and passes the matched value through the function passed in to the Parser's bind function. This gives us a Result

Monad, and we pull out it's value (if it exists).
- Then we build the new Result Monad holding `(matched value, remainder of text)`, except now the matched value is the result of `function`.

This lets us bind functions that affect the result we get at the end of our parsing, by changing the match, but not interrupting the parsing, by changing the remainder of the text.

```
def fmap(self, function):
    return self.bind(lambda x: Result.ok(function(x)))
```

We're also going to introduce a new function very similar to `bind` called `fmap`. This function is the same as `bind`, but it takes a function that can't fail and uses that to modify the matched value. We could always write out `bind(lambda x: Result.ok(function(x)))` for this, but it's nice to have it packaged up in a method, because then we can bind it to a symbol.

```
# >>
def __rshift__(self, function):
    return self.bind(function)

# >
def __gt__(self, function):
    return self.fmap(function)
```

Finally, let's write some functions that actually call the parser on some strings.

```
def parse_prefix(self, string):
    return self(string)

def parse_total(self, string):

    def check_full(tup):
        if tup[1] == '':
            return Result.ok(tup[0])
        else:
            return Result.error('The match did not consist of the entire '
                + 'string: {} was left over'.format(repr(tup[1])))

    return self(string) >> check_full
```

These two functions help make it easier to use the parser; instead of calling the parser ourself, we use these functions to parse either a prefix of the string using our parser, or the entire string.

Using the Parser Combinator

Now we have a powerful enough Parser Monad to recreate all of the flexibility and power of regular expressions. We can combine parsers to make more complex ones, we can define functions that return parsers that recreate common regular expressions syntax, so let's try parsing something simple; let's try and write a Parser Combinator that parses numbers.

A number can be as simple as `12` or as complex as `12.00123e45`, so we're going to need to build up a complex parser. Let's start with creating a parser that parses 1 or more of consecutive digits.

```
digits = Parser.oneof('0123456789').many1()
```

Now, we need to express an optional decimal place, followed by one of more digits.

```
decimal = (Parser.char('.') + digits).optional()
```

Now, we need an exponent part, which is pretty simple given the above. However, we do need one additional component, a sign.

```
sign = (Parser.char('+') | Parser.char('-')).optional()
exponent = ((Parser.char('e') | Parser.char('E')) + sign + digits).optional()
```

Finally, we can put these three together to get:

```
number = sign + digits + decimal + exponent
```

Here's our finished code, and it's results on some possible inputs:

```
digits = Parser.oneof('0123456789').many1()
decimal = (Parser.char('.') + digits).optional()
sign = (Parser.char('+') | Parser.char('-')).optional()
exponent = ((Parser.char('e') | Parser.char('E')) + sign + digits).optional()
number = sign + digits + decimal + exponent

Parser.parse_total(number, '12')
    # Ok('12')
Parser.parse_total(number, '12e10')
    # Ok('12e10')
Parser.parse_total(number, '2.12345e100')
    # Ok('2.12345e100')
Parser.parse_total(number, 'hello world')
    # Error(Failed to match one of
    # ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'])
Parser.parse_total(number, '99 bottles of beer on the wall')
    # Error(The match did not consist of the entire string:
    # ' bottles of beer on the wall' was left over)
```

```
Parser.parse_total(number, '')
    # Error(End of String encountered)
```

Furthermore, we can use a result-oriented version of python's `float` function, along with our `bind` operator, to cast these results to be floats instead of strings.

```
def result_float(x):
    try:
        return Result.ok(float(x))
    except Exception:
        return Result.error('Failed to cast to a float')

Parser.parse_total(number, '7.22345e10') >> result_float
    # Ok(72234500000.0)
```

One really

Going Further

Everything we've done so far can basically be done in the exact same way by regular expressions. However, because of the way we've defined the Parsing Combinator, we can use `bind` and `fmap` inside of Parsing Combinator expressions. This allows us to parse complex expressions into fully formed and entirely arbitrary python objects.

```
number = (sign + digits + decimal + exponent) >> result_float
```

As above, we can put the cast to a float inside of the number parser, and it will automatically do the conversion whenever something is parsed. We can also take this further:

```
whitespace = Parser.oneof(' \n').many1()

# without infix operators
many_numbers = (
    (whitespace.optional().last(number)).fmap(lambda x: [x])
    ).many()

# with infix operators
many_numbers = (
        (whitespace.optional() >= number) > (lambda x: [x])
    ).many()
```

Let me take this text and expand it into English. This reads 'If there's any whitespace, match it and discard it, and parse a number following it. Then, put that number in a list, and repeat until parsing fails. Then combine all of the lists (via concatenation), and return that.' And this works perfectly, 100% in

pure standard python, with no `eval` or `exec` statements. And you absolutely cannot do this with regular expressions.

```
text = '2.12345e+100 2.1e10 1223 13.5 100e100'
print(Parser.parse_total(many_numbers, text))
# Ok([2.12345e+100, 21000000000.0, 1223.0, 13.5, 1e+102])
```

Parsing Combinators can be used to write powerful, modular, readable, and concise parsers for any format of text. For example, here is a CSV parser in 5 lines of code.

```
expression = Parser.noneof(',\n').many1()
comma = Parser.char(',')
newline = Parser.char('\n')
line = ((expression <= comma.optional()) > lambda x: [x]).many()
csv = ((line <= newline.optional()) > (lambda x: [x])).many()

text = '''1,2,3,4,5
hello world, my, good, friends, 5
0,1,2,3,4'''

print(csv.parse_total(text))
# Ok(
#     [
#         ['1', '2', '3', '4', '5'],
#         ['hello world', ' my', ' good', ' friends', ' 5'],
#         ['0', '1', '2', '3', '4']
#     ]
# )
```

And finally, here is a full arithmetic expression parser. This is more powerful than the power of regular expressions to break a string into tokens. It constructs the full AST from a purely textual input, just using a Parser Combinator. You'll notice that at this point, I need to define a function and pass that in to a Parser; we need to do this because this is a recursive expression we are parsing. However, it doesn't actually add to the complexity too much.

```
from collections import namedtuple
from enum import Enum, auto

class Op(Enum):
    PLUS = auto()
    MINUS = auto()
    TIMES = auto()
    DIV = auto()

Expr = namedtuple('Expr', ['Op','e1','e2'])
```

```
openp = Parser.char('(') + whitespace.optional()
closep = whitespace.optional() + Parser.char(')')
plus = whitespace.optional() + Parser.char('+') + whitespace.optional()
minus = whitespace.optional() + Parser.char('-') + whitespace.optional()
times = whitespace.optional() + Parser.char('*') + whitespace.optional()
div = whitespace.optional() + Parser.char('/') + whitespace.optional()

Plus = lambda x: Expr(Op.PLUS, x[0], x[1])
Minus = lambda x: Expr(Op.MINUS, x[0], x[1])
Times = lambda x: Expr(Op.TIMES, x[0], x[1])
Div = lambda x: Expr(Op.DIV, x[0], x[1])

def expr(text):
    # rec stands for recursive
    rec_plus = ((openp >= Parser(expr)) & (plus >= Parser(expr))) <= closep
    rec_minus = ((openp >= Parser(expr)) & (minus >= Parser(expr))) <= closep
    rec_times = ((openp >= Parser(expr)) & (times >= Parser(expr))) <= closep
    rec_div = ((openp >= Parser(expr)) & (div >= Parser(expr))) <= closep

    full = (
                (rec_plus > Plus)
                | (rec_minus > Minus)
                | (rec_times > Times)
                | (rec_div > Div)
                | number
           )

    return full(text)

text = '((1+2) * (9 - 11))'

print(Parser(expr).parse_total(text))
# Ok(
#     Expr(
#         Op=<Op.TIMES: 3>,
#         e1=Expr(
#             Op=<Op.PLUS: 1>,
#             e1=1.0,
#             e2=2.0
#         ),
#         e2=Expr(
#             Op=<Op.MINUS: 2>,
#             e1=9.0,
#             e2=11.0
#         )
#     )
```

```
# )
```

Theory Time

Now that we've seen many types of Monads, we should discuss the technical, boring discussion of what a Monad is. A Monad is, in essence, any construct, usually an object, in a language that satisfies the following criteria. I will say each criterion in two ways; a simple way, and a correct way.

However, there are two ways to define Monads. The two definitions in practice lead to the exact same object, but some Monads are easier to deal with using one definition, and other Monads are easier to deal with using the other. I will start with the Monad laws that are easiest for the Option Monad.

The Monad Laws with `bind`

- For any type `t`, and Monad type `M`, `M t` is the type of the Monad holding that type.
- *Monads can hold values*

- There is a function called 'unit' or 'return' that has type `t -> M t` which injects a value of type `t` into a Monad of type `M t` in some simple way.
- *There is a function to create Monads from regular values*

- There is a binding operation of type `M t -> (t -> M u) -> M u` which maps the value of a Monad of type `M t` into another Monad of type `M u` by using a function that maps a value of type `t` into a Monad of type `M u`
- *There is a bind function, as discussed earlier*

These above laws also have to follow a set of rules that determines how the above functions can be combined. These rules are super simple; basically they just exist to make sure that the bind function and the constructor don't do anything funky.

- The unit function acts as a neutral element of bind
- *Calling bind on the unit function (or constructor) doesn't do anything*

Example:

```
Option.some(5).bind(Option.some) == Option.some(5)
```

- Binding two functions in series is the same as binding the result of composing those two functions
- *You don't have to worry about binding doing weird things to your values; an example is really useful here*

```
def one_over(x):
    if x == 0:
         return Option.none()
    return Option.some(1/x)

def two_over(x):
    if x == 0:
        return Option.none()
    return Option.some(2/x)

Option.some(5).bind(one_over).bind(two_over) == Option.some(5) \
    .bind(lambda x: one_over(x).bind(two_over))
```

`fmap` and `join`

If you want, you can replace the bind function with two slightly different functions, and you get the same exact system. These two functions are called `fmap` and `join`. Below, I'll implement them for the Option Monad.

```
    def fmap(self, function):
        if self.is_none():
            return self
        val = self.unwrap()
        return Option.some(function(val))

    def join(self):
        if self.is_none():
            return self
        val = self.unwrap()
        return val
```

`fmap` is the simpler of the two. `fmap` applies a function to the Option's value, if it has one. The difference between `fmap` and `bind` is that `fmap` assumes that it's function will always succeed, and therefore doesn't need to return an Option Monad. We have already seen `fmap`, with the Parsing Combinator, but we haven't really given it it's full introduction until now.

```
def plus_one(x):
    return x + 1
```

```
Option.some(5).fmap(plus_one)
```

In the above example, we can't use `bind`, because `plus_one` doesn't return an Option Monad, so it would break our chain of `bind` commands. However, `fmap` can be used instead. But `fmap` can't be a replacement for `bind` all on it's own.

```
def one_over(x):
    if x == 0:
         return Option.none()
    return Option.some(1/x)

Option.some(5).fmap(one_over) == Option.some(Option.some(0.2))
Option.some(0).fmap(one_over) == Option.some(Option.none())
```

In the above example, `fmap` adds an additional layer of the Option Monad. This is where `join` comes in. `join` collapses two layers of a Monad into one layer.

```
Option.some(5).fmap(one_over).join() == Option.some(0.2)
Option.some(0).fmap(one_over).join() == Option.none()
Option.none().fmap(one_over).join() == Option.none()
```

As a matter of practice, most monads will implement all three; `bind`, `fmap`, and `join`. Annoyingly, different languages and libraries name these functions different things, but they're probably there under different names.

The Monad Laws with `fmap` and `join`

A lot of these laws are the exact same.

- For any type `t`, and Monad type `M`, `M t` is the type of the Monad holding that type.
- *Monads can hold values*

- There is a function called 'unit' or 'return' that has type `t -> M t` which injects a value of type `t` into a Monad of type `M t` in some simple way.
- *There is a function to create Monads from regular values*

- There is a mapping operation of type `M t -> (t -> u) -> M u` which maps the value of a Monad of type `M t` into another Monad of type `M u` by using a function that maps a value of type `t` into a Monad of type `M u`
- *There is an fmap function, as discussed earlier*

- There is a mapping operation of type `M M t ->  M t` which
- *There is an fmap function, as discussed earlier*

These above laws also have to follow a set of rules that determines how the above functions can be combined. These rules are very similar to the `bind` rules.

- The identity function acts as the neutral element of fmap
- *Calling fmap on the function f(x) = x doesn't do anything*

Example:

```
Option.some(5).fmap(lambda x: x) == Option.some(5)
```

- The unit function acts as the neutral element of the composition of fmap and join.
- *Join doesn't do anything weird to values.*

Example:

```
Option.some(5).fmap(Option.some).join() == Option.some(5)
```

- Fmapping two functions in series is the same as Fmapping the result of composing those two functions.
- *You don't have to worry about fmapping doing weird things to your values; an example is really useful here*

```
def plus_one(x):
    return x + 1

def plus_two(x):
    return x + 2

Option.some(5).fmap(plus_one).fmap(plus_two) == Option.some(5) \
    .bind(lambda x: plus_two(plus_one(x)))
```

The Zeroth Monad

Our first section was titled 'Our First Monad'. However, we are computer scientists, and therefore we start counting at zero, not at one. So let's talk about another Monad that everyone reading this document has probably used, but never noticed that it was a monad. A list.

How is a list a Monad? Well, from the previous section, a Monad is really just anything with a `bind` function, or with a `fmap` and a `join` function. And while

not every programming language has these functions built in, we can easily write these functions for a list.

```
# python has a built in function, 'map' that does this.
def fmap(ls, function):
    new = []
    for item in ls:
        new.append(function(item))
    return new

# this is sometimes called 'flatten'
def join(ls):
    new = []
    for sublist in ls:
        for item in sublist:
            new.append(item)
    return new

def bind(ls, function):
    return join(fmap(ls, function))
```

This is all nice and well that we now have these functions, but it explains little conceptually. So let's try and describe Monads conceptually, and see how that can be applied to lists.

Our first Monads, the Option Monad and the Result Monad, both represented some sort of computation result that required more context than a simple value; in particular, they represented a computation result that could either succeed, producing a value, or fail, producing no meaningful value.

Lists can be thought of in a similar way; instead of representing either zero or one meaningful return value, lists can represent computations that can return zero, one, or any possible number of return values. For example, consider the following contrived example.

```
def less_than_abs(x):
    if x == 0:
        return []
    ls = [0]
    for i in range(1,x):
        ls.append(i)
        ls.append(-i)
    return ls

from math import sqrt

def sqrts(x):
    if x < 0:
```

```
        return []
    elif x == 0:
        return [0.0]
    else:
        return [sqrt(x), -sqrt(x)]

bind(less_than_abs(3), sqrts)
# [0.0, 1.0, -1.0, 1.4142135623730951, -1.4142135623730951]
```

In this case, we execute two functions in series, getting all of the valid results to our question in one list; but the number of results isn't the same for all inputs, so we need a Monad to represent this computational uncertainty.